Visor++: A Visualisation Tool for Concurrent Object-Oriented Programs

Michael J. Oudshoorn[1]
Hendra Widjaja[2]
University of Adelaide

Abstract

The use of program visualisation for understanding and fine-tuning task-parallel object-oriented programs is desirable. One reason is that such programs typically involve complex interactions between the program entities. Combined with other tools, program visualisation tools can make understanding and fine-tuning of such programs easier. For maximum benefit, the visualisation of task-parallel object-oriented programs should not focus only on a small section of program features, but rather on a wide cross section which covers all the important program interactions. The resulting views should be presented in ways that facilitate ease of use, ease of relating one view to another, and should assist users in building a mental model of the program. In other words, a holistic approach to visualisation is required.

This paper discusses Visor++, a tool for visualising CC++ programs. This tool embodies the concepts of a holistic approach to present views which are integrated and inter-related. Although the underlying system of CC++ does not support program visualisation, it is indeed possible to devise such visualisation at the language level. The usefulness of Visor++ is demonstrated by some representative cases.

Keywords: software visualisation, visualisation, concurrent, object-oriented, CC++.

[1] Departmnt of Computer Science, University of Adelaide, Adelaide, SA, 5005, Australia. E-mail: michaelcs.adelaide.edu.au

[2] Current address: Internet & Interactive Media Group, National Computer Systems, Pte Ltd., 81 Science Park Drive, #04–3/04, The Chadwick, Singapore 118257. E-mail: whendra@ncs.com.sg

1 INTRODUCTION

Concurrent object-oriented programming is a powerful paradigm in that it allows the modelling of real-world entities as objects which interact concurrently. However, understanding the execution of concurrent object-oriented programs can be difficult. This is due to the fact that the execution of such programs may involve a complex interaction of abstract objects which can be difficult to fathom [3]. Program visualisation can help alleviate this problem.

Program visualisation is simply defined as the use of graphical artifacts to enhance the understanding of programs [9, 15, 16]. In a wider context, it is sometimes also called *software visualisation* [15]. Through program visualisation, the operations of a program are represented by graphical icons. Interactions among program entities are then represented as interactions among those icons. Program visualisation can be done at the abstract algorithmic level, or at the language level. In any case, program visualisation can be used to help users, especially programmers, to more easily relate the program execution to the original mental model of the program [10, 12, 14].

This paper explores many aspects of visualising concurrent object-oriented programs, in particular task-parallel object-oriented programs. It is partly motivated by the fact that there are relatively few visualisation tools for concurrent object-oriented programs, particularly for task-parallel object-oriented programs, and that the views provided by these tools vary. Furthermore, most such visualisation tools provide visualisation by using the information obtained at the underlying systems level. In other words, the visualisation is driven by program run-time data obtained at this level. This requires changes to the operating systems, the language run-time system, or the compiler. This paper explores the possibility of providing visualisation at the program level. The paper also focuses on investigating the types of views suitable for visualising task-parallel object-oriented programs. Furthermore, investigation is also carried out to determine how far such an approach allows the provision of meaningful visualisation.

For experimentation, the CC++ language [7] is used as the target of visualisation. The framework for visualising such programs is realised by the tool Visor++, which utilises Polka [17]. Polka itself is a graphical package which is specially tailored to provide visualisation of concurrent programs.

This paper describes the approach taken to provide such program visualisation. Section 2 discusses some related work in the visualisation of concurrent object-oriented programs.

Section 3 highlights the rationale, architecture and the views provided by Visor++. Some experimentation with Visor++ and analysis thereof are covered in Section 4, followed by a discussion in Section 5. Concluding remarks are given in Section 6.

2 RELATED WORK

There are relatively few examples of the application of program visualisation to the concurrent object-oriented paradigm. One possible reason is that the paradigm is relatively new. Some of these visualisation tools are discussed below.

MVD (Monitoring, Visualisation and Debugging) [5] is a set of tools for visualising the execution of μC++ [4] programs. μC++ itself is a concurrent language, extending C++ with four basic abstractions: coroutines, monitors, coroutine-monitors, and tasks. The visualisation support that MVD provides focuses on these abstractions, to the extent that only these abstractions and their interactions are visualised. Other entities, such as object member function invocations, are not included. μC++ programs are visualised by using *trace visualisation*, which is visualisation based on program execution traces. MVD also supports *statistical visualisation*, in which statistics of program execution is displayed, such as task activity status, and stack high-water marks.

The TAU [2, 13] visualisation tools are part of the pC++ programming language system [1]. pC++ itself is an extension of C++, with several additional constructs to support data parallelism. The TAU tools are implemented as a graphical hypertool, in which it is a composition of several tools, each of which supports unique capabilities. The tools are divided into two categories. The first category consists of static analysis tools, including **fancy** for browsing global functions and class methods, **cagey** for displaying the static program call graph, and **classy** for displaying the static class hierarchy. The second category comprises dynamic analysis tools, such as **racy** for displaying usage profiles of functions and concurrent object member functions, **easy** for displaying program events on an X-Y graph, and **breezy**, a breakpoint-based debugger. Similar to μC++, the pC++ system provides the necessary support to instrument and obtain trace information from its programs.

The LAMINA program visualisation tool is used to visualise the execution of LAMINA [8] programs. This tool provides a number of views which can be used for performance debugging. The **network-operator view** displays the processor-network view, along with the load on each processor, and communication among them. Latency and utilisation on each processor are also displayed. Another view, the **activity table**, displays textually the activities of each concurrent object, such as the number of messages that have been processed, and the average execution time.

Another example is the visualisation system for PARC++ [18] programs. Similar to μC++ and pC++, this language is derived from C++. It extends C++ with the necessary abstractions to support concurrency, such as thread management, thread communication, and thread synchronisation. PARC++ programs can be visualised by using several tools, thereby providing different angles of visualisation. Some important program elements, however, do not seem to be visualised in the views. For example, the tool **Visit** [18] does not provide views of monitors, and the tool POPAI [18] does not visualise thread executions, while both monitors and threads are important concepts in PARC++. Visualisation in PARC++ does not provide views of static program elements either, thereby making it difficult for users to meaningfully compose their mental models.

All the tools cited above display language-level views, that is, the views deal with language-level entities, in which interactions among those entities are represented as icons. The tools can also use statistical views for determining program performance. The reason that such views are employed is clear: they assist programmers in understanding programs, and pinpointing performance problems. However, there are three difficulties.

Firstly, many such tools provide views which are loosely inter-related. Different entities in different views may represent different concepts. Consistent coding can alleviate the problem of linking the visual entities with the mental model of the program. However, when the number of such entities exceeds the capabilities of the human short-term memory to cope with, explicit support must be used. Such support should make it easier for users to interpret the meanings of those entities and their colours, and how they explicitly relate to the source code.

Secondly, many of the tools provide *specialist support* for program understanding and fine-tuning. For example, some tools specialise in visualising message passing and communication in message-passing programs. Some other tools specialise in visualising threads only. Although these are viable approaches, this paper examines a more holistic approach: a visualisation tool for program understanding and fine-tuning should take into account not only specific program sections or features, but also the other features that may play role in determining the flow of program execution.

Thirdly, many such tools as represented above use language support for providing information for driving program visualisation. For example, μC++ and PARC++ were specifically created by incorporating features for easily extracting data for visualisation. Therefore, they are "visualisation-conscious" [13]. Other languages, such as LAMINA, do not have such support. Nevertheless, their underlying run-time systems permit easy incorporation of such support. However, the majority of concurrent object-oriented languages are not visualisation-conscious [13]. Visualisation is usually an after-thought. This is one difficulty which must be addressed.

With the above problems in mind, the specific goals of this paper are three-fold. Firstly, to examine the types of views suitable for visualising task-parallel object-oriented programs. The visualisation is targeted for use by programmers for understanding programs and tuning their performance. Secondly, to examine ways such that the entities in the views are strongly related to each other and to the source code. Thirdly, to examine mechanisms which permit languages that are not visualisation-conscious to be able to support visualisation, and to examine to what extent such support can be provided.

A framework called Visor++ is introduced for visualising CC++ programs. CC++ is ideal as an experimental vehicle, for it is a task-parallel language which is not visualisation-conscious.

3 VISOR++

To appreciate the issues involved in the design of Visor++, a brief overview of CC++ is given prior to the discussion of Visor++.

3.1 CC++ and Visualisation

The CC++ language is a declarative, task-parallel language, which is a strict superset of C++. The extensions to C++ allow the construction of parallel programs from simpler components through the use of sequential, concurrent, and parallel composition. To create a CC++ program, the best sequential algorithms can be employed for each component, which can later be composed to form a single concurrent, or parallel program.

A CC++ program consists of one or more *processor objects* (POs) which reside on one or more physical processors. A PO is basically an abstraction of locality, in which each such object has a separate address space. In turn, on each PO, one or more threads can execute. In terms of POs, a CC++ program is composed of one or more POs, which in turn is composed of one or more threads. CC++ provides constructs for synchronisation among threads, and constructs for RPCs among entities (for example: threads) in different POs.

3.2 Design Considerations

In designing the views of Visor++, two main considerations must be taken into account. Firstly, which language entities should be the focus for visualisation, and secondly, what views can be devised to appropriately visualise them?

3.2.1 Entities

In providing holistic program visualisation, the choice of which language features to visualise are important. In Visor++, the language features chosen are those which are relatively coarsely-grained which have relatively significant impacts on program execution. In particular, the entities are threads, functions, logical processor objects, and remote procedure calls. Other entities, such as variables and program data structures are ignored. This does not mean that they are unimportant. The reason, rather, is that in task-parallel object-oriented programs, it is the tasks and their interactions which are of utmost importance. Variables and data structures are, perhaps, better dealt with by a visual debugger or by tools dealing with the visualisation of data-parallel object-oriented programs. Thread synchronisation constructs, on the other hand, are visually implicit in thread views. As a result, a wide cross-section of language entities are used, without sacrificing the levels of details that can be visualised.

3.2.2 Views

As described in Section 1, program visualisation can be used to help users, especially programmers, to more easily relate program execution to the original mental model conceived of a program [10, 12, 14]. This, then, makes it possible for the programmers to understand the differences between the original program specification and the real behaviour of a program. Visor++ assists in building mental model in terms of the original source code.

To realise the above point, in Visor++, all views are linked back to the source code, whenever possible. There are two components that form such links: *static views* and *dynamic links*. Static views refer to the views depicting static program information, including the **source-code view**, and the **class-hierarchy view**. Dynamic links, on the other hand, depict the relationships between *dynamic views* and the static views. In Visor++, dynamic views are the views depicting the dynamic operations of the programs in terms of the visualisation entities (see Section 3.2.1).

In short, the views in Visor++ consist of static and dynamic views, in which the entities inside the dynamic views may refer to the corresponding entities in the static views. The reference can be to *declaration points* and *invocation points*. Therefore, a programmer can find, for example, where a function is declared, and where the function is invoked from.

Static and dynamic views are described in more detail below.

1. **Static views.** Static views are the views of the static properties of a program. They include the source-code view, and the class hierarchy view. Both views are shown in the right hand side of Figure 1.

2. **Dynamic views.** These views depict dynamic program execution. To make the use of the views more manageable, these dynamic views are arranged in a hierarchical manner. This means that lower-level views are hidden and can be displayed upon request. Currently, there are only two levels of views:

   (a) **Global views.** These views depict the global (computation-wide) operations or status of a program execution. They include the following views:

   - **Processor- and processor-object-related views.** These display the statistics, such as idle time and computation time of processor and processor-objects.
   - **RPC-related views.** These views display the RPC activities occuring in a CC++ program, including the call statistics.
   - **Function usage views.** These views display the frequency of invocations for all the functions computation-wide. The views can be useful for optimising the functions which are most often used, thereby shortening overall computation time.

   (b) **Local views.** Local views depict the computation at the processor-object level. In Visor++, these views are not displayed until the user specifically requests them. Local views are comprised of the two views described below:

   - **Thread view.** The thread view displays the threads which are active at each particular point in time. This view is essentially a space-time diagram, in which the vertical axis represents the threads, and the horizontal axis represents time. Each thread is displayed as a thin bar extending to the right, as execution time advances. In this view, each thread is displayed as a sequence of function invocations. Each segment within a thread bar represents the invocation of a particular function, in which each function is represented by a distinctive colour.
   - **Function stack view.** This view is similar to the thread view, except that it shows the stack of function calls being made by each thread. Since this view and the thread view are closely related, they are grouped together as two sub-windows within a single window,

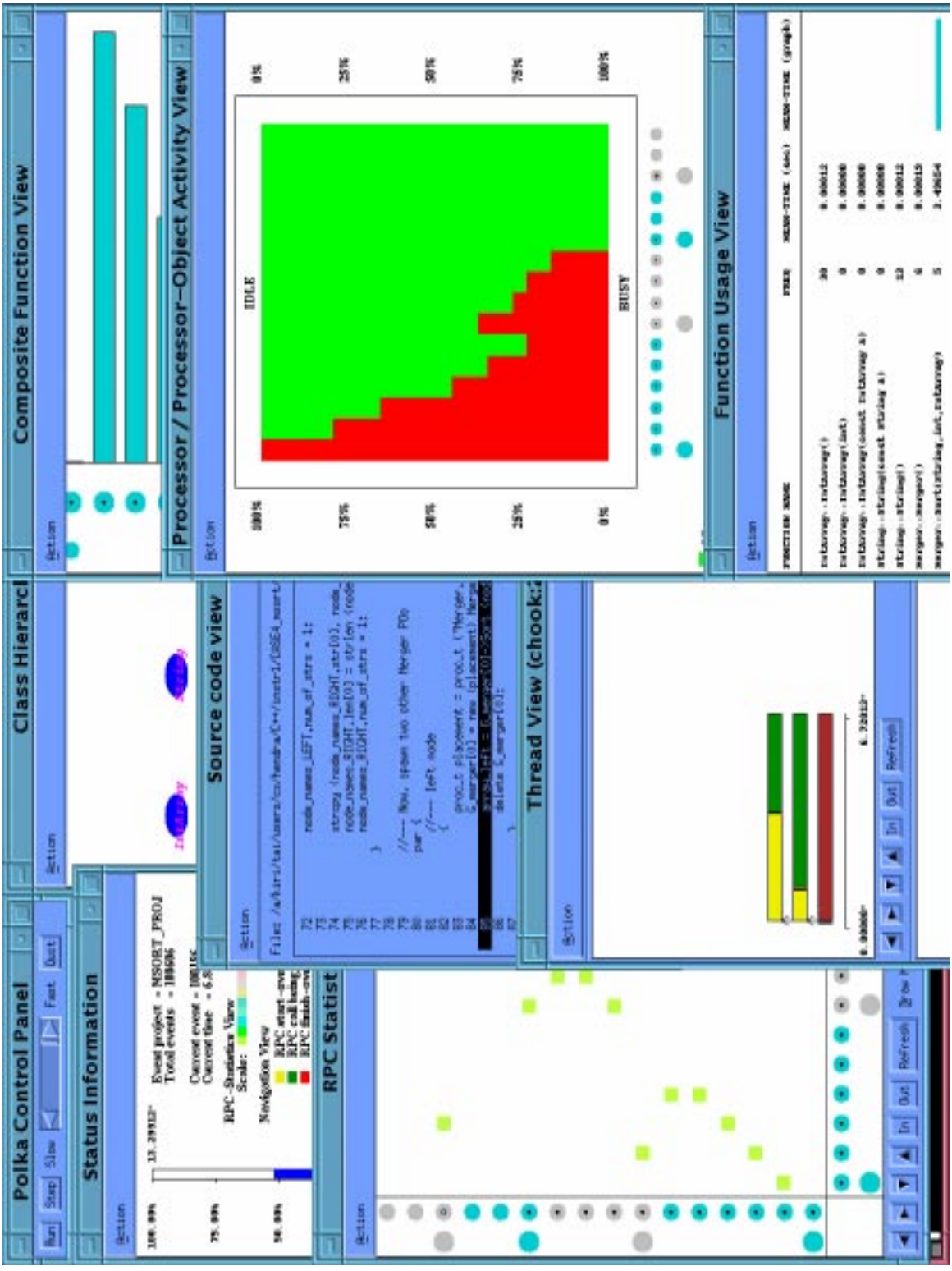

Figure 1: *Visor++ in execution.*

Figure 2: *Auxiliary view showing information of a function.*

with the thread view at the top position. Figure 1 shows only the thread view portion of this window.

It can be seen that the views in Visor++ are arranged in a hierarchical way. The invocation or display of lower-level views is activated only upon the user's request. This means that lower-level views can be displayed or hidden accordingly.

3. **Auxiliary views.**

   To make the views more strongly related, another kind of view, the *A-view* (auxiliary view) is used. This view has two purposes. Firstly, they can "remind" users of what a particular display entity stands for. Secondly, they can serve as a vehicle to link the entities in different views representing the same concept. For example, Figure 2 is an A-view which is displayed when the user selects an entity in the function view.

Visor++ is implemented with the above concepts in mind.

3.3 Visor++ Design

Visor++ is essentially composed of three subsystems, as shown in Figure 3. Due to space limitations, only a brief description is given below. A more complete description can be found in [20].

The static and dynamic views are produced and driven by the *event visualisation subsystem* by using the visualisation tool POLKA. The static views are created by using the static program information obtained during the static analysis of a program. The dynamic views, however, are created and driven based on the run-time data representing program states during execution. This data is collected during program run-time, and consolidated into a database. Therefore, Visor++ is a post-mortem program visualisation system.

The key of producing the static and the dynamic views lie on the program static analysis and program instrumentation phases, respectively. Before a program is executed, it is first statically analysed to determine its static structures, such as program file structures, and class hierarchies. After static analysis, the program is instrumented by inserting "probes" inside the original program. These probes, together with the original program, will emit data that reflects program states during execution.

It should be noted that both the static analysis and the program instrumentation are carried out at the source-code level only. Furthermore, both are done automatically by the system. This is particularly useful, since the user is freed from the tedious and error-prone process of manually carrying them out. Such an approach follows the one used in TAU. The approach is effective for two reasons. Firstly, it results in a highly portable system which does not rely on the intricacies of the underlying system. Secondly, it can indeed be used to produce the views as outlined previously.

In the next section, three experiments are conducted to highlight the merits of the framework of Visor++.

4 USING VISOR++

The first experiment is the visualisation of a small and simple distributed master-slave program adapted from the tutorial for CC++ [6]. The program involves one master processor object (PO) spawning one or more slave POs, each of which sends a message to be printed back to the master PO and then terminates. This is a very simple program consisting only of RPCs. Visor++ is used to visualise their RPC activities. The RPC activity view and the thread view reveal that a considerable amount of RPC time is spent on the creation of POs. This can be explained by using the fact that the creation of a PO means using RPCs to create a separate execution address space, possibly on a different physical processor. The same views also reveal that RPCs, in general, are very expensive in terms of execution time. Although this example is very simple, it illustrates how the usage of Visor++ can reveal important facts about a program, hence helping programmers to better understand their codes. In the next two examples, it will be demonstrated how using Visor++ views can help programmers *both* to understand and to fine-tune their programs.

The second experiment is the optimisation of a distributed merge-sort program, also adapted from [6]. The program uses a master-slave configuration in the form of a binary tree of processor-objects (POs). The internal nodes are the *merge-POs* which implement the merge operation, while the leaves are the *sort-POs* implementing the sorting operation. Each merge-PO divides the data it receives into two equal halves, spawns two other merge-POs (or two sort-POs at the leaves of the tree), merges the results, and passes them back to its parent PO.

An analysis of a merge-sort program using a 4-level binary tree is conducted with Visor++. Using Visor++, the program is automatically instrumented, and the execution traces of the instrumented program is visualised. Initially, each merge-PO is placed on a different physical processor (see the left part of Figure 4), using 7 different machines. The rationale is that as each merge-PO is computationally expensive, it needs to be placed on a separate processor. Each sort-PO, however, is placed on the same processor as its parent. The rationale is that since a UNIX time-sharing system is being used, as many processor cycles as possible should be used for sorting, while at the same time reducing the amount of RPCs. Using Visor++, the program is automatically instrumented, and the execution traces of the instrumented program are visualised.

During visualisation, the processor activity view reveals that the POs do not have much idle time (Figure 1 is, in fact, a snapshot of Visor++ when visualising this program). However, the utilisation of the physical processors is rather poor. As in the first example, by using the RPC activity view and the thread views, and the link-back to the corresponding source-code, it is found that synchronous RPCs dominate the computation. While a merge-PO is performing two RPCs to its slave merge-POs (or sort-POs), it is effectively blocked, hence being idle. This observation leads to the idea of placing those POs into less physical nodes to

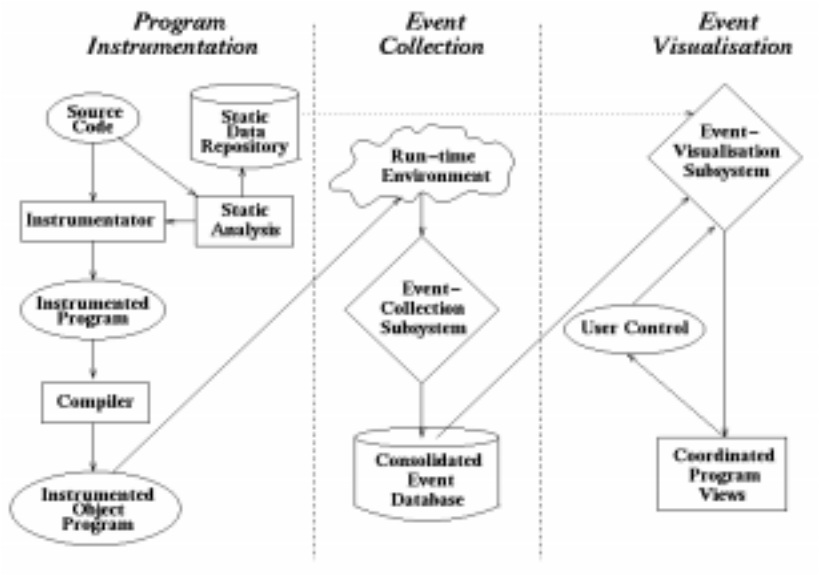

Figure 3: *General framework of Visor++.*

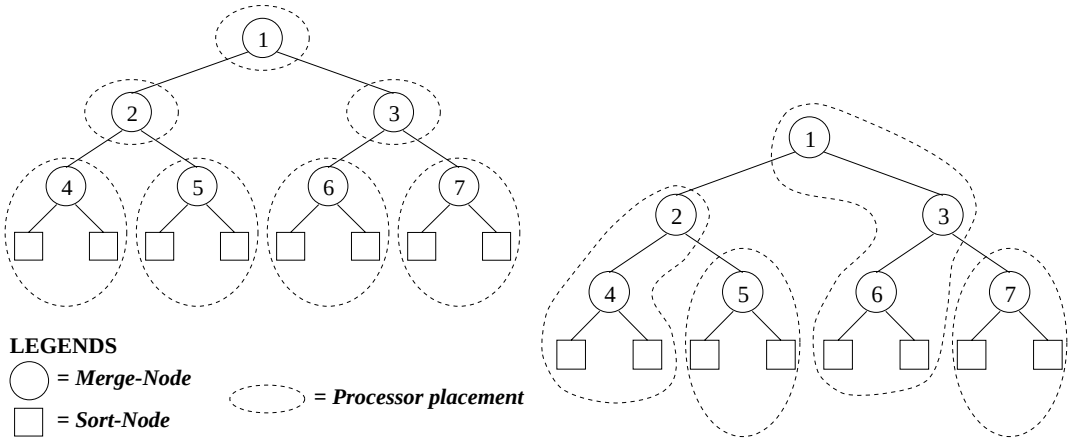

Figure 4: *Merge-sort using a 4-level binary tree. The left figure shows the initial placement of the nodes on processors, and the right figure the optimised placement.*

	Un-instrumented	Instrumented	% change
Original placement	18.057 secs	26.515 secs	46.8 %
Final placement	9.723 secs	11.981 secs	23.2 %
Improvement	185.70 %	221.31 %	–

Table 1: Timing information from the merge-sort program.

increase efficiency. The final placement is shown on the right side of Figure 4.

Table 1[2] provides the timing information for sorting about 20,000 integers. The program with the initial PO placement takes approximately 18 seconds to execute. With the alternative placement, execution time is reduced to 10 seconds while using less processors. The column "instrumented" gives the measurements of the instrumented versions of the same programs. Further optimisations are, of course, possible. For example, the function view can be used to highlight which functions are most frequently invoked or are longest to execute, and optimise accordingly. However, it is the intention of the paper to show that optimisation is possible by cleverly arranging computation structures, even without changing a single line of code. This insight is achieved by using Visor++

The final example is the visualisation of a parallel text-searching program, which has been described in more detail elsewhere [20]. The text-searching program searches for a string text among a set of N text files. The final output is a list of L files in which the string is found, where $0 \leq L \leq N$. In the implementation, P processor-objects (POs) are used, in which $N \ rem \ P$ of them are assigned $\lceil N/P \rceil$ files each, and each of the remainder is assigned $\lfloor N/P \rfloor$ files. The program is implemented as a branch-and-bound algorithm, in which a master-PO allocates and places slave-POs on separate physical processors. The master-PO then initiates RPCs to obtain results from the slave POs. To do this, the master-PO creates P threads, each of which allocates, places, and performs RPCs to a slave-PO.

Using Visor++ views, an optimisation loophole is pinpointed. The views show that the master-PO is idle while the slave-POs are executing. Therefore, the master-PO could be modified so that it participates in the search. After modification, using Visor++, it is found that in some cases, although L files have been found by some slave-POs, yet many other slave-POs continue executing the search. To optimise, the program is modified so that when the master-PO detects that it already has a list of L files found, it instructs the slave-POs to stop computing. The results of this experiment is presented in Table 2, where $L = 3$, $N = 20$, $P = 3$, and the text string to be found is "biochemistry".

5 DISCUSSION

The results in the previous section indicate that when properly used and interpreted, the views in Visor++ can help users to understand and fine-tune their programs. By using the views, once program understanding is achieved, fine-tuning can be done. Such tuning may or may not involve modification of source code, as previously demonstrated. Consequently, a wide variety of programs can be analysed in this way by using Visor++.

Using Visor++, many program facets can be uncovered. This is possible, since Visor++ incorporates a holistic approach, i.e. incorporating a wide cross-section of program features for visualisation. The views can be *vertically-expanded* to include higher-level algorithmic views, and lower-level system views.

The experiments with Visor++ also uncover the fact that program instrumentation produces *probe effects* [11], which means that program execution and its flow are affected by the insertion of the visualisation probes. This can be seen from the two tables in Section 4. The exact effects of such probes depend on the program being instrumented. However, some measures have been taken in Visor++ to minimise such effects. Observations have indicated that the additional amount of time needed by an instrumented program as opposed to its un-instrumented version varies consistently between 10 % to 47 %. This is still acceptable, taking into account that the resulting gain in execution time can be potentially much more than such perturbations. Furthermore, after executing visualisation and subsequently effecting necessary changes to a program, the probes can be removed.

The experiments also have indicated that using the current implementation, Visor++ is effective only for small to medium-sized programs (about 5000 lines of code). The reason is that large programs, when visualised, tend to have a large number of entities displayed. A user or programmer may be submerged by such a myriad of information. Under such circumstances, a tool should have more intuitive and more sophisticated visual cues to guide the user throughout visualisation. Vertival extensions, as previously mentioned, can also be used. Furthermore, the incorporated views must have more *spatial* and *semantic immediacy* [19]. This means that the views should be spatially arranged in such a way that the interpretation or linkage of one view agains the others should be made easier. More visual cues to help the user to link the views semantically to the original program is also needed. Given the complex nature of task-parallel object-oriented programs, such extensions are yet to be examined.

6 SUMMARY AND CONCLUSIONS

This paper presents a framework for visualising the execution of task-parallel object-oriented programs. This framework entails some important concepts. Firstly, the views are inter-related through the use of view structuring, consistent colour coding, and explicit referral to the static code structures. Secondly, program visualisation for program understanding and fine-tuning should be based on a holistic approach in that a fairly wide section of important program features should be used for visualisation. Presented in reasonable ways, the visualisation of these features assist users to better understand and fine-tune programs. Finally, language-level support for visualisation is indispensable, as it allows a tight integration between the language and its visualisation tools. Such support should be automated as far as possible.

The above framework has been embodied in the tool Visor++ to visualise CC++ programs. Several case studies presented demonstrate the usefulness of the concepts. Properly used, these concepts could help users to better understand and pin-point problems. This framework, however, can still be improved, particularly for visualising large programs. These extensions include, among others, the vertical extension of the views to include both lower-level views and higher-level algorithmic views, which provide stronger spatial and semantic immediacy to the user. More sophisticated visual cues and intelligent agents can also be used. Given the complex nature of task-parallel object-oriented programs, however, these issues remain to be solved.

Despite the potentials of program visualisation, it is indeed no panacea to the problem of program understanding and fine-tuning. Rather, program visualisation can be used in conjunction with other tools for these purposes. A proper coupling between program visualisation tools and

[2] To level out the spikes in machine and network loads, the original program, the un-instrumented and the instrumented versions, are executed 10 times in an interleaved fashion, and the averages were taken. The measurements for the improved program are carried out likewise.

	Un-instrumented	Instrumented	% change
Original program	28.076 secs	36.634 secs	30.5 %
Improved program	12.436 secs	14.359 secs	15.5 %
Improvement	225.77 %	255.13 %	–

Table 2: Timing information from the parallel text-searching programs.

other kinds of tools is potentially beneficial.

REFERENCES

[1] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of the 1993 Supercomputing Conference, Portland, Oregon*, pages 588–597, 1993.

[2] D. Brown, S. Hackstadt, A. Malony and A. Malony. Program Analysis Environments for Parallel Language Systems: The TAU Environment. In *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing, Townsend, Tennessee, USA*, pages 162–171, 1994.

[3] M.H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, Volume 18, Number 3, pages 177–186, July 1984.

[4] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger and C.R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software-Practice and Experience*, Volume 22, Number 2, pages 137–172, February 1992.

[5] P.A. Buhr and M. Karsten. *μC++ Monitoring, Visualisation and Debugging, Annotated Reference Manual, Preliminary Draft.* Department of Computer Science, University of Waterloo, Waterloo, Canada, version 1.0 edition, March 1996.

[6] CC++ Designer Team. *CC++ Tutorial.* Department of Computer Science, California Insitute of Technology, Pasadena, California, 1994.

[7] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In G. Agha, P. Wegner and A. Yonezawa (editors), *Research Directions in Concurrent Object-Oriented Programming*, Chapter 11, pages 282–313. The MIT Press, Cambridge, Massachusetts, 1993.

[8] B.A. Delagi, N.P. Saraiya and S. Nishimura. Monitoring Concurrent Object-Based Programs. In G. Agha, P. Wegner and A. Yonezawa (editors), *Research Directions in Concurrent Object-Oriented Programming*, Chapter 15, pages 479–509. The MIT Press, Cambridge, Massachusetts, 1993.

[9] S. Ellershaw and M.J. Oudshoorn. Program Visualisation — The State of the Art. Technical Report TR94-19, Department of Computer Science, University of Adelaide, November 1994.

[10] V. Fix, S. Wiedenbeck and J. Scholtz. Mental Representations of Programs by Novices and Experts. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel and T. White (editors), *Proceedings of the Conference on Human Factors in Computing Systems, INTERACT'93 and CHI'93, The Netherlands*, pages 74–79. ACM, April 1993.

[11] E. Kraemer and J.T. Stasko. The Visualisation of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, Volume 18, pages 105–117, 1993.

[12] P. Lyons, C. Simmons and M. Apperley. Hyperpascal: A Visual Language to Model Idea Space. In *Proceedings of the 13th New Zealand Computer Society Conference*, pages 492–508, New Zealand, August 1993.

[13] B. Mohr, D. Brown and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In B. Buchberger and J. Volkert (editors), *Lecture Notes in Computer Science volume 854, Proceedings of the International Conference on Vector and Parallel Processing, CONPAR'94*, pages 29–40. Springer-Verlag; Berlin, Germany, 1994.

[14] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, Volume 19, pages 295–341, 1987.

[15] B.A. Price, R.M. Baecker and I.S. Small. A Principled Taxonomy of Software Visualisation. *Journal of Visual Languages and Computing*, Volume 4, Number 3, pages 211–266, September 1993.

[16] G. Roman and K.C. Cox. Program Visualisation: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia*, pages 412–420, May 1992.

[17] J.T. Stasko and E. Kraemer. A Methodology for Building Application-Specific Visualisations of Parallel Programs. *Journal of Parallel and Distributed Computing*, Volume 18, pages 258–264, 1993.

[18] K. Todter and C. Hammer. PARC++: A Parallel C++. *Software Practice and Experience*, Volume 25, Number 6, pages 623–636, June 1995.

[19] D. Ungar, H. Lieberman and C. Fry. Debugging and the Experience of Immediacy. *Communications of the ACM*, Volume 40, Number 4, pages 38–43, April 1997.

[20] H. Widjaja and M.J. Oudshoorn. Concurrent Object-Oriented Programming — A Visualisation Challenge. In *Proceedings of the Conference of Visual Data Exploration and Analysis IV, IS&T/SPIE Symposium on Electronic Imaging: Science and Technology, San Jose, California, February 1997*, pages 310–321, San Jose, California, February 1997.